

Henri Levälampi

Modular Web Development Framework

Helsinki Metropolia University of Applied Sciences

Bachelor of Engineering

Information and Communication Technology

Thesis

4 December, 2015

Author(s)	Henri Levälampi
Title	Modular Web Development Framework
Number of Pages	48 pages
Date	4 December, 2015
Degree	Bachelor of Engineering
Degree Programme	Information and Communication Technology
Specialisation option	Software Engineering
Instructor(s)	Erja Nikunen, Principal Lecturer Ali Abdul-Karim, CEO, Marimo Games
<p>This thesis studies how to make a great performing development environment to create great performing websites. The result of this thesis is a set of tools and conventions that can be thought of as a framework - a base to build new web applications with. The conventions are gathered from personal experiences and research on each given topic. The purpose of this study was to discover methods for creating static webpages with a sophisticated templating system.</p> <p>This thesis was carried out by developing a site for a pilot project and mapping the workflow and the tools used to create the site. The framework consists of open source libraries and self-developed components. The development workflow is enhanced with task automation tools that reduce the amount of redundant work by providing the ability to use pre-processors and on-the-fly optimization to generate the distribution files.</p> <p>A website is not built only with development tools - the content of the site needs to be structured and designed so it can be viewed in different kinds of end devices. The study gives a brief overview on the general principles of design - visual clues and laws of grouping. These help to make the information on the website more absorbable for the user. Web visibility is also discussed by going through search engine guidelines.</p> <p>One of the big influencing factors in website user experience is speed. The study discusses the ways to optimize distribution especially on mobile networks to provide a engaging experience for the users.</p> <p>The framework produced by this study has proven itself in practice to be a worthy base a projects and a working environment for website projects. The framework will be further developed and possibly used in customer project after this current study.</p>	
Keywords	Website optimization, website usability, web development workflow, gulp.js, npm, Sass, handlebars.js

Tekijä(t)	Henri Levälampi
Otsikko	Modulaarinen web-kehitysrunko
Sivumäärä	48 sivua
Aika	04.12.2015
Tutkinto	Insinööri (AMK)
Koulutusohjelma	Tietotekniikka
Suuntautumisvaihtoehto	Ohjelmistotuotanto
Ohjaaja(t)	Erja Nikunen, Yliopettaja Ali Abdul-Karim, Toimitusjohtaja, Marimo Games
<p>Työn tarkoituksena oli tutkia ja löytää tekniikoita ja tapoja tuottaa suorituskykyisiä ja käyttäjäystävällisiä verkkosivustoja. Tarkoituksena oli luoda uudelleenkäytettävä web-kehitysrunko, jota voisi käyttää uusien projektien pohjana. Esitetyt työtavat ja tekniikat valittiin henkilökohtaisten kokemusten ja tutkimustyön tuloksena. Työssä keskityttiin staattisten web-sivustojen kehittämiseen.</p> <p>Työn kehitysrunko muodostui samalla, kun pilottiprojektia kehitettiin. Kehitysrunko muodostuu lähinnä avoimen lähdekoodin kirjastoista ja työkaluista sekä joistain itsekehitettyistä komponenteista. Kehitysrungossa on automaatiotyökaluja, jotka helpottavat kehittäjän työtä automatisoimalla turhaa, itseään toistavaa työtä tarjoamalla mahdollisuuden käyttää esikääntäjiä ja jakelutiedostojen optimointia lennosta.</p> <p>Web-sivuja ei rakenneta pelkillä kehitystyökaluilla - sivuston sisältö täytyy suunnitella rakenteeltaan ja tyyleiltään helposti omaksuttavaan muotoon ja toimimaan erilaisilla päätelaitteilla. Työ käy lyhyesti läpi suunnittelun peruseriaatteita kuten ryhmittelysääntöjä ja käyttäjän huomion ohjaamista kontrastia luomalla. Työssä käydään myös läpi hakukonenäkyvyyteen vaikuttavia seikkoja.</p> <p>Yksi merkittävämmistä käytettävyyteen vaikuttavista ominaisuuksista on sivuston nopeus - työssä käydään läpi web-sivun optimointia varsinkin mobiiliverkkojen osalta.</p> <p>Lopputyönä syntynyt web-kehitysrunko on osoittautunut toimivaksi tavaksi kehittää verkkosivuja ja sen edelleenkehitys jatkuu.</p>	
Avainsanat	Verkkosivun optimointi, verkkosivun käytettävyys, web-kehityksen työnkulku, gulp.js, npm, Sass, handlebars.js

Contents

Abbreviations

1	Introduction	2
1.1	Purpose	3
1.2	Structure	4
2	General Principles of Website Design	5
2.1	Laws of Grouping	6
2.2	Visual Clues	8
2.3	Layout Structure	8
3	User Experience on Web	11
3.1	Content Relevancy	11
3.2	Usability	12
3.3	Distribution Optimization	13
3.4	Mobile Browsing	14
3.5	Responsive Design	16
3.6	Browser variation	19
3.7	Critical Rendering Path	20
3.8	Semantics and Search Engine Optimization	24
4	Framework Development Technologies	26
4.1	Templating with Mustache and Handlebars	26
4.2	Styling with Sass	29
4.3	Optimizing Images	30
4.4	Modularity with Package Managing	33
4.5	JavaScript Scoping Convention	34
4.6	Back-end	35
5	Developer's Workflow with Framework	37
5.1	Framework structure	37
5.2	Version Control System	38
5.3	Browser Developer Tools	40
5.4	Task Automatization with Gulp	41
6	Summary	44
	References	46

Abbreviations

API	Application Programming Interface. Provides an interface to use pre-defined functions from an application.
break-point	Code debugger mark for pausing code execution.
cache	Place to store files to improve performance.
CDN	Content Delivery Network. A large system of servers to improve performance in file distribution.
CMS	Content Management System. Application for creating, editing and publishing content.
cookie	Small piece of data stored in the browser.
CSS	Cascading Style Sheets. Used to style markup languages.
CSSOM	CSS Object Model. CSS styles mapped into a tree-like model.
DOM	Document Object Model. Convention for represent and interacting with objects in HTML.
ES6	ECMAScript 6 or ECMAScript 2015. Standardized specification for JavaScript language.
Facebook	Social networking service.
fps	Frames per second.
Git	Version control system.
GitHub	Web service for using Git.
Gulp	Toolkit for task automation.
gzip	Data compression file format. Used to compress content when transferring it in web.
Handlebars	Templating language extended from Mustache.
HTML	Hypertext Markup Language. Markup language to create webpages.

HTTP	Hypertext Transfer Protocol. Request-response protocol for transferring hypertext.
JPG	Joint Photographic Experts Group. Lossy file format for digital images.
JSON	JavaScript Object Notation. Human-readable data format for storing objects as attribute-value pairs.
LinkedIn	Social business networking service.
mixin	Set of declarations that can be re-used in other rulesets by calling the mixin.
Node	Event-driven server-side web application environment. Applications are written in JavaScript.
PNG	Portable Network Graphics. Lossless file format for digital images.
public domain	Registered identifying string for a online service.
Sass	Syntactically Awesome Stylesheets. Adds functionalities to CSS.
scope	Current context in the code.
SCSS	Sassy CSS. Block-type markup for Sass.
Slack	Instant messenger application.
social media	Online services to share content and information with individuals and virtual communities.
SVG	Scalable Vector Graphics. XML based file format that can hold vector graphics, raster graphics and text.
Twitter	Micro-blogging service.
url slug	Part of a website url.
viewport	Area where content is rendered in a browser.
web presence	Individual or company has content on the Internet.

1 Introduction

The Internet is a powerful tool for delivering information. Individuals and companies can offer information for all to view and use. Information can be put to third party services such as Facebook or LinkedIn, or content can be hosted from self made site through registered *domain name*. Hosting an own website gives more freedom on what and how to show the content, whereas the third party solutions offer a more strict but easier way to share information to the public.

When an individual or a company has content in the Internet they have a *web presence*. From the company's stand point to have good web presence means that the company has a solid website but can also reach users and be visible in the services their *target group* uses. Usually these services are *social media* networks such as Twitter and Facebook. Posts in social media help to have insight to the company values but social media also acts as an easy way to reach companies without writing a formal mail. The company could also have a blog where to showcase its knowledge and skills by sharing helpful and informative articles.

Web presence can be a crucial factor when people are looking up products and services from the Internet using *search engine* services such as Google. A service or product with a good web presence is more likely to be chosen. When the site structure is done correctly search engines give it more points and it gets higher in search results. And when the website is well put together, it gives a convincing first impression. Be it a company, individual or an open source framework, a good web presence gives an edge in a competition - it is like going to a job interview with an appropriate and stylish look. [11]

1.1 Purpose

The purpose of this thesis is to introduce a way to develop websites, an agile environment with tools and conventions to create well performing static websites with the help of automation tools. The introduced way of working can be thought of as a framework as it is a collection of tools that have been implemented to work together to produce something new. The goal of the framework is to ease the development process by reducing redundant work and providing a ready thought out project structure. The framework introduced in this thesis has a modular structure and thus can be easily customized for different kinds of projects.

This work goes through website design and usability principles and then introduces a workflow and development environment to create an engaging user experience. The study gives an understanding how to establish or enhance a web presence with a well functioning website. The introduced framework is useful for creating static websites with effective, yet simple, templating system with variables and some basic flow controlling statements e.g. `if` and `for` loop. With variables and flow control, a single template markup can be used to produce different results. Because of modular structure of the framework, dynamic content can also be added with a help of a *back-end*. Back-end refers to a service that receives the user requests and processes a custom response based on the user input.

In practice building the framework means gathering tools and conventions to be used in future projects. The tools used consist from *open source* libraries and self developed components and configurations. These tools and conventions are chosen based on the author's studies and experience from working on client projects.

To discover the conventions and tools needed for creating a website, one must be developed. Marimo Games's site needed a makeover as there was more new content to be shown on the site. The development environment could be built and the thesis could be written around process. This environment would take shape while developing the pilot project for Marimo Games.

Marimo Games is a Finnish game company founded in the fall of 2014. The company's focus is currently in mobile games development. At the time of writing this thesis, Marimo Games had two employees.

Since Marimo Games's first game was planned to be released in Summer 2015 there was about four months to redesign and develop the website to get a proper web presence for the company. The product delivered for the Marimo Games is a website built with the framework. The framework itself is not built for Marimo Games. The deadline for the website was June 2015 as the first game from the Marimo Games was planned to be released in following months. One of the main goals was to not have a back-end for the Marimo Games's site as the static site can handle more traffic.

1.2 Structure

The study is divided to chapters by development phases - the most focus being in the user experience design, development environment setup and framework technologies. Each phase is introduced and discussed. There are also examples on how each development phase was implemented into the framework and how it was utilized in the pilot project.

In this document *framework* refers the tools and conventions gathered for web development environment, *user* refers to website visitor and *developer* to a person working with the framework. Italics are used to emphasize an important key word.

2 General Principles of Website Design

This chapter briefly goes through the most important objective aspects of the website design as they influence the way the site is implemented in the code- and markup-level. Design work itself cannot be automated and the framework does not provide tools for the design phase. Usually pen and paper are the best anyway.

A website designing task is not purely artistic - *visual variables* and *principles* must be taken into consideration. These visual clues help the human brain to process the information presented. Figure 1 demonstrates the use of visual clues on text. The content in the text blocks, *example 1* and *2*, are the same but *example 2* has some additional styling.

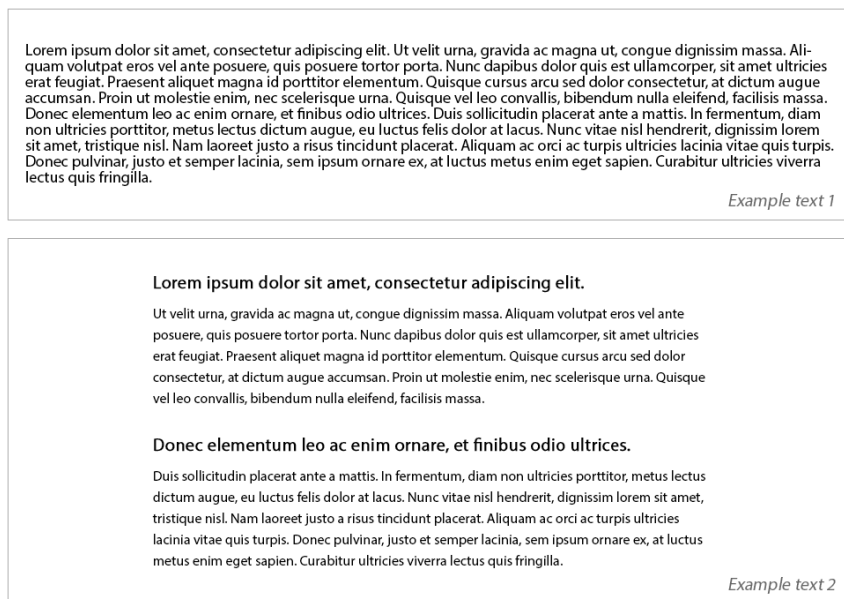


Figure 1: Example texts 1 and 2 have the same exact text content.

Text elements are separated from each other with margin and two sentences are styled as titles above body text elements. The title predicates what the paragraphs under it contain. Titles are made more important in relation to paragraphs by placing them above the related paragraphs and styling title text differently from the normal body text. The same elements should be styled identically or nearly identically as the consistency helps

to perceive the relations between elements. The relations can be expressed with *size*, *proximity* and *similarity*. Additionally appropriate *line height* and *column width* improve readability. Also fonts on the site should be picked carefully as the text content is read with it. [17]

2.1 Laws of Grouping

Design aesthetics are subjective but complexity is not, as the visual variables and principles are based on usability and human mind studies. *Gestalt Principles* came up multiple times during the research for this objective design topic. Gestalt Principles explain how the human mind structures and organizes visual information which are represented by the laws of grouping: *proximity*, *similarity*, *closure*, *symmetry*, *continuity*, *figure-ground*. The human mind tends to look at things as a whole and not as a sum of their parts - Gestalt Principles explain how. These kinds of studies help developers to create more intuitive interfaces and layouts where the information is more absorbable. Figure 2 and 3 shows proximity and similarity as in the laws of grouping. [13]

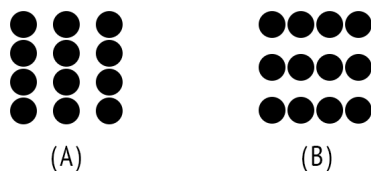


Figure 2: Proximity

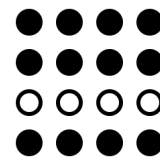


Figure 3: Similarity

Objects closer together appear to be grouped and related as seen in Figure 2. The balls in group A and B are positioned differently. In group A, balls appear in rows and in group B, balls appear in columns. Also similarly styled objects appear grouped as seen in Figure 3. Proximity and similarity can be used to express relativeness of items without a need for heavier visual clues such as separator lines or borders.

When shapes overlap, the human mind separates the visual field as figure (foreground) and ground (background) as Figure 4 demonstrates. The foreground (square) has the primary attention and the background (circle) is more passive. Figure-ground can be used to create icons for the interface. Icons usually take less space and are more intuitive than text buttons. [13]



Figure 4: Figure-ground

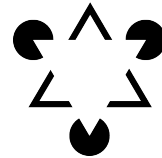


Figure 5: Closure

Incomplete objects are perceived as whole objects rather than separate pieces as demonstrated in Figure 5. The image is seen as there would be a white triangle put on top of dots and a triangle shaped frame, and not like there are six separate shapes in a circular pattern. Human vision tends to see continuous forms, rather than pieces of it (see figure 6). [13]



Figure 6: Continuity

Continuous form is simpler to process than different separate shapes. Brain adds the missing information if necessary. Human brain tries to perceive complex scenes by reducing complexity and seeing simple symmetrical shapes as the figure 7 illustrates. [13]

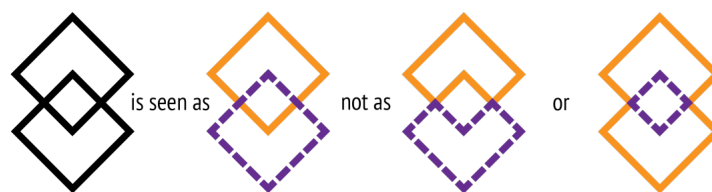


Figure 7: Symmetry

As a summary, human mind perceives the visual information as a whole and usually in its simplest possible form. The whole carries a different meaning than the parts of it alone. The knowledge gathered on how human brain perceives information is the key for making intuitive and efficient layouts and interfaces.

2.2 Visual Clues

Visual clues help guide users to what is important. The variables used in the clues are color, size, shape, position, orientation and movement. With these variables *contrast* can be created - contrast is what catches the users attention.

Different variable values should be used to indicate different functionalities. For example with text, different colors can indicate different states and interactions: passive, highlight, clickable, disabled (see Figure 8). This only works if the use of visual variables is consistent and the more sparingly contrast is used, the more effective it becomes.



Figure 8: Common visual variables used with text.

As some visual variables are used coherently across websites and user interfaces, it is easier to understand their functions as presented in Figure 8. The most dominantly used text color does not usually have interaction. Highlighted, differently colored text pops and if it is underlined it is usually clickable. Faded texts are usually disabled and cannot be interacted for some reason. Interaction can be emphasized and indicated with different cursor types when hovering over the text.

2.3 Layout Structure

When the content wanted to be shown in the website is gathered, modules needed to show the content can be determined. From there the basic *wireframe* of the website can be drawn. Wireframe is usually a sketch with abstract content elements and shapes - no actual text or images are usually used (see Figure 9 on the next page). Wireframe is created to help determine where to put each content module before designing the visuals of them. Content should determine the structure of the site. Wireframe is created to ensure that layout is logical and the information in it is processed as wanted by the users - what to show first and what to hide behind a link.

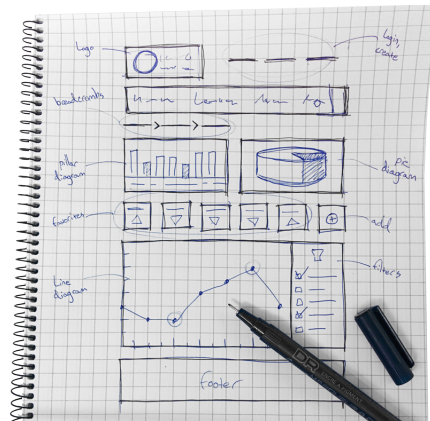


Figure 9: Example of a wireframe sketch.

Designing a web page differs from other forms of document designing as a web page is not just a static sheet of paper. A web page has to be designed so it looks and feels good on various devices and screen sizes. This aspect of design is called *responsive design*. The flow and order of the content must be thought so it works on all devices. This is usually achieved by using *columns* or *grid* based layout. On wider viewport more content can be shown with horizontal columns. When viewport gets narrower, the number of columns is reduced and the content is shown more vertically (see Figure 10). The flow of the layout is determined by setting maximum and minimum portions for each element in each layout option.

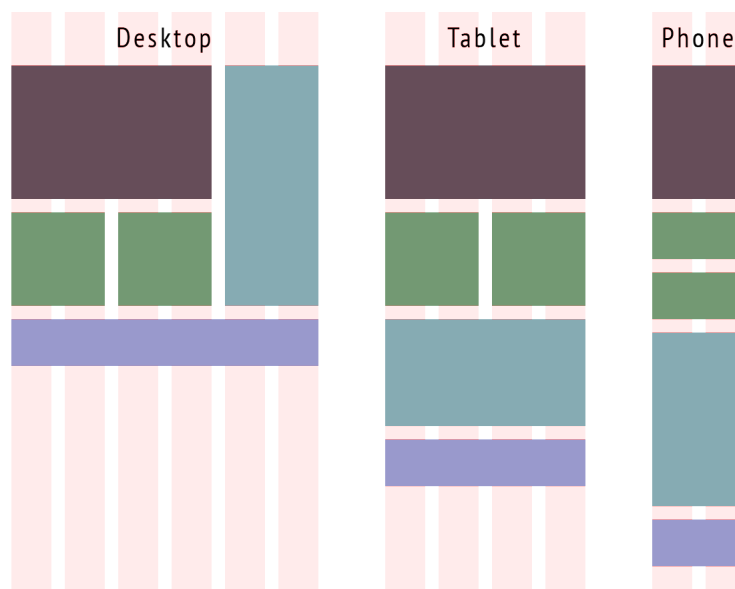


Figure 10: Grid based layouts for desktop, tablet and phone screens.

It is often good to respect the common site structure that the users are familiar with. Site structure is achieved by separating content into sections. Separation can be done with borders, colors or space. If there is a lot of content, it may be appropriate to hide some of

the content with menus and pagination. With a small amount of content, it is unnecessary to split it into separate pages or menu items as this is only inconvenient for the user.

To see if layout is well structured and highlights the wanted elements, one technique is to look at the layout from a distance or blur it and notice if the wanted parts still stand out (see Figure 11).

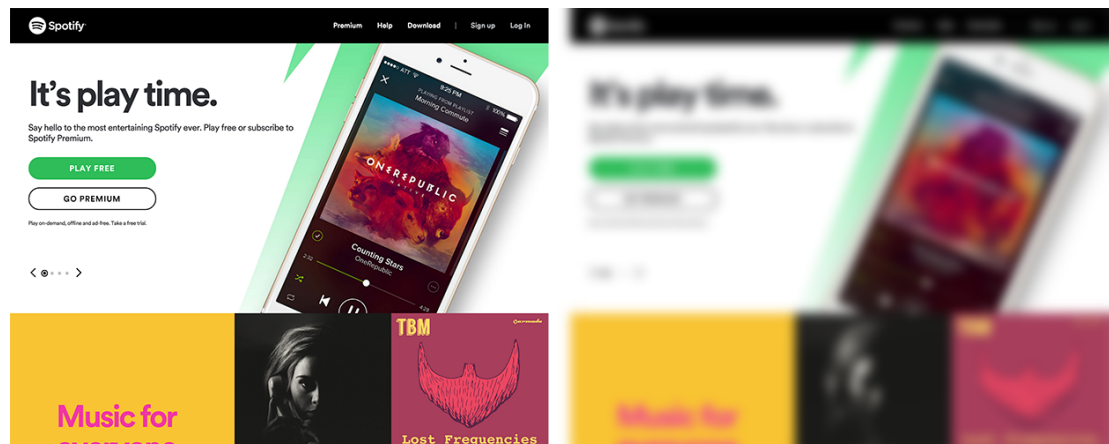


Figure 11: www.spotify.com/uk as normal on the left and blurred on the right.

From the blurred image attention focuses first to the picture of a phone and song cover art as they are big and colorful. Also the “Play free” button grasps some attention as the contrast with its background is high. At the same time smaller details such as top menu items and carousel pagination spheres fade out.

3 User Experience on Web

This chapter covers more technical design aspects of building a web service. These apply on a generic level, not only for this framework created in the present study.

There are standards written by *The World Wide Web Consortium (W3C)* in *Web Content Accessibility Guidelines (WCAG) 2.0* for making the web content as accessible for everyone as possible. Another well known guideline is the Nielsen Norman Group's *Ten Usability Heuristics*. The article about these were written in 1995 - they are as relevant today as they were years ago. [19] [25]

The experience of using a website consists of its content and the way it is presented and distributed. Quality of presentation includes that the browser is able to render elements quickly and smoothly. Distribution has many different layers for optimization as the server configuration, the user's browser and the device its running on, the possible back-end of the application influence greatly the result. To achieve good overall performance for the website, it requires insight on how networks and browsers work.

3.1 Content Relevancy

When arriving to a website users should be able to recognize that they have come to the right place. Content relevancy is highly subjective and situational, for example a weather forecast site can be expected to contain information about the upcoming weather and not about how to raise a dog. Additionally the content should be up-to-date, for example a year old weather forecast is not relevant for a average user visiting weather page when they are looking tomorrow's weather. But for a user gathering statistics for global warming study, the year old forecast is relevant. Content is relevant if a user finds it valuable.

If a user arriving to the site does not find the content relevant or the site is too slow, the user will immediately leave. This is called *bouncing*, as users navigate out from the site as quickly as they came.

3.2 Usability

One of the key usability aspects is speed. For a user, under 100 milliseconds delay feels instant. 100 to 300 millisecond delay is notable. 300 to 1000 milliseconds makes the user feel that the machine is processing given input. Over one second delay will likely cause a mental context-switch. [11]

All user actions should receive instant feedback. For example when a form is sent by clicking the submit button, the interface should instantly indicate that the button press is registered, the form is being submitted and give the user instructions to wait for a confirmation that the form is received successfully. If the interface does not indicate the submit action has occurred, user may get confused and click the submit button multiple times. This might cause an error in the server side and if user is not advised to wait, user might leave while not been informed about unsuccessful submit.

Interface should not require instructions how to use it - this does not mean there should not be any, but rather that the interface should be intuitive. For example in a password form field it is better to guide the user about the password character requirements beforehand, rather than leave it up to the users to figure out through trial and error.

Visible information should be structured in such a way that the user is not overwhelmed. There is a fine balance on how much content is good to show at once. With a vast amount of content it is also good to let the user determine how many, in what order and in what listing layout items are shown. By following the common site structure, users have easier time navigating the site as it feels familiar.

3.3 Distribution Optimization

Before a user can view and use the page, its files must be loaded to the users device. To get these files to user as efficiently as possible the files need to be optimized. Fortunately this has been partly automated in the framework by using *Gulp* and its plugins.

Data travels long distances through cable. There are many types of cables such as copper and optical fiber. Currently the fastest data can travel is nearly at the speed of light in the optical fiber cable. So as the data is not instantly jumping from a place to another, there will always be *latency*. Latency is the time it takes the source to send a packet to the destination. [8] [11]

Content Delivery Network (CDN) can help to reduce latency as it can provide the resources from geologically nearest source for the user. Browser limits the active connection to a certain amount with a single domain. With CDN operating from a different domain than the main site, this gives more active connections to load resources in parallel. This is why loading resources from multiple domains gives more throughput. [8]

Another limitation in the network is the bandwidth. Bandwidth refers to the maximum throughput of a logical or physical communication path. The maximum throughput of a bandwidth is determined by the lowest performing link in the network pipeline between sender and receiver. The figure 12 illustrates how the bandwidth is affected by the steps between the end points.

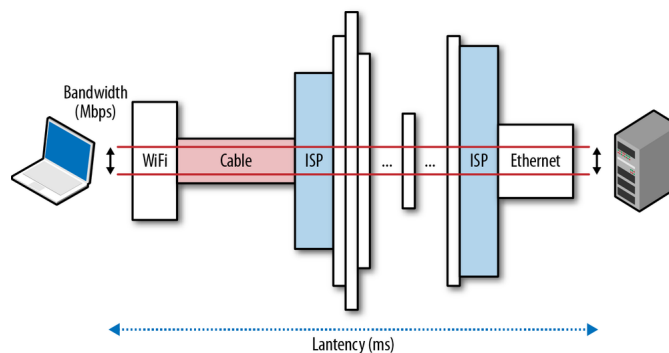


Figure 12: Bandwidth and latency illustrated between the client and server. [8]

For example a user could have a 100 Mbit/s network plan from the *Internet service provider* (ISP), but if user's router is only capable for 50Mbit/s throughput, the real-world performance is capped to the weakest link. In this situation it would be the 50Mbit/s.

When developing the web site, the site markup and code is preferably in a human-readable form. But when the browser renders the markup and code, it does not have to be. So to save network *bandwidth* and reduce the file sizes, the files should be *minimized*. Line breaks, comments and unnecessary spaces are removed in the minimization process. It is also possible to minimize code so that the variable and function names are converted into single character aliases.

After the resource is minimized it should be compressed before serving it to users. Compression means encoding the data into fewer bits and thus reducing the file size. *GZIP* is used for text-based assets such as HTML, CSS and JavaScript. The compression rate is usually 70%-90%. GZIP is configured on server-side. Images are optimized with compression algorithms and removing unnecessary meta data. Each file type has its own set of minimization methods. [8] [11]

3.4 Mobile Browsing

As the mobile browsing is getting more popular, sites should be made mobile-friendly. When looking at the data from *StatCounter Global Stats* [24], at the time of writing this thesis (November 2015) about 40% of the world's Internet traffic was generated with a mobile device. In Europe the mobile percentage is a little lower, around 30%.

A mobile browser does not just mean a smartphone browsers but rather that the whole user is mobile. The end-device can be something between a smartwatch and a laptop. Mobile devices usually have lower performance than desktops but even if mobile devices could perform as well as desktop computers, they have a limited source of power - a battery. It is alright to produce more interesting content with animations and effects but as these enhancements require heavier processing from the end device, this will drain

the battery more quickly. So that in mind, web applications that are used hours in one session, user will appreciate the optimized performance.

When optimizing web page for a mobile user it comes to compromising between quality and speed as mobile networks are usually limited compared to static wired connections. Compared to wired connections, mobile data transferred between the user and server is usually traveling longer road through service provider's radio towers and internal network. As there are more steps between the end points, the delay is usually longer. This delay is also referred as *latency*. On the mobile networks the latencies are notably greater compared to wired connections. Figure 13 describes latencies with different connection types.

Generation	Data rate	Latency
2G	100–400 Kbit/s	300–1000 ms
3G	0.5–5 Mbit/s	100–500 ms
4G	1–50 Mbit/s	< 100 ms

Figure 13: Data rate and latency for active connection in different mobile network generations. [8]

On top of these *active connection* latencies there is a handshake delay. The handshake has to be done so the connection can be established between the service providers radio tower and the user's device. This can take around 10 to 100 milliseconds with newer mobile network generations and with older, sub-4G networks, this can take several seconds. When inspecting the usability delay threshold ranges, these mobile network latencies do not leave much time for the actual website file loading and processing before the user gets frustrated. [8]

Compared to cable connections, interference is a bigger problem with wireless connections, as the data is transferred through air. The actual *code rate* in wireless connections are lower due to stronger *error correction*. Error correction ensures that the data won't distort when its being transported. A downside for the error correction is that it takes up

bandwidth from the actual data that is wanted to be transferred. The more interference, the more error correction and the lower the code rate to carry meaningful data. [11] [2]

As the websites are developed with a desktop or a laptop, the performance aspect for mobile networks and hardware are easily omitted. Luckily emulators offer a way for developers to test out the site with unstable connection. Google's *Chrome browser* has one built-in on the *developer tools* and *OSX* has a *Network Link Conditioner* (see Figure 14) that can be loaded as a *Xcode* plugin.

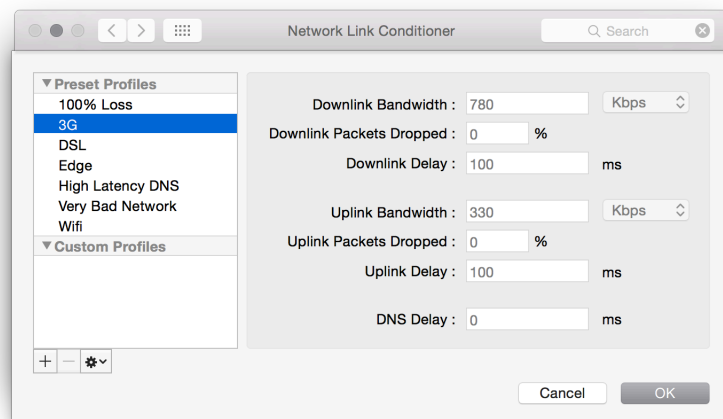


Figure 14: OSX Network Link Conditioner preferences. Network profile set to 3G.

The impression that these tools provide should be considered as a directional clue. There are no guaranteed data rates in the real world as there are so many factors that impact the mobile network performance. A good general goal would be to have less than 100 requests and under one megabyte of data transferred when the page is loaded and ready to be used.

3.5 Responsive Design

Most important tools in responsive design are *viewport* settings, CSS media query *break-points* and *relative units*. With these the layout can be implemented to adapt to the users device. Currently smallest notable screen size in use is in older iPhone's - iPhone 5 and older. Those devices have very narrow screen of 320 pixels. Thus the 320 pixel width can be set as a design limit.

Content should be the same for all devices. A good way to approach the responsiveness is to think it as an *incremental enhancement*. So rather than taking away content and features when screen size and performance drops, this should be thought another way around. The starting point is the most low-end device, and with wider screens and better hardware, content is made more richer by adding effects and by expanding the content by using the available space in the screen.

When mobile devices arrived with their mobile browsers, most of the websites were not designed to be viewed with smaller screens than desktop monitors. So mobile browser vendors made the browsers scale the content down so the whole page fitted into the smaller screens. This required users to zoom in and out to navigate the site. Gladly, mobile users are currently considered and users are not required to zoom to read the content. To achieve this on smaller screens, one of the most important settings to configure is the *viewport*. [16]

```
1 <meta name="viewport" content="width=device-width, initial-scale=1">
```

Listing 1: Viewport meta tag.

Viewport is the virtual area where the browser renders the content. To configure the viewport settings to be ideal for mobile devices, in the viewport meta tag the *width* is set to *device-width* (see Listing 1). By telling this in the viewport tag, the browser knows not to zoom out the content in the browsers viewport (see Figure 15). *Width* in the viewport tag present the virtual viewport of the device - the area that user views the content from the browser. *Device-width* represents the actual physical device screen width. [16]



Figure 15: Viewport test page with emulated mobile device (iPhone 4s). On the left with appropriate viewport value and on the right with no viewport set.

Breakpoints are used to activate styles on certain circumstances. CSS *media queries* can be set to *media types* and to *media features*. Media type is used to recognize different types of use cases. For example *screen* type identifies devices with a normal display such as smartphones, tablets or desktops and when printing a page, the device type is set to *print*. [15]

Most commonly used media features would be the viewport size, orientation and pixel ratio. Most of the media features accept prefixes *min* and *max* to replace the usage of 'less-than' and 'greater-than' (< and >) characters as they may conflict with the HTML and XML markup. These prefixes give a way to define ranges in feature value to trigger styles. For example with a wider screen, the site layout could be divided into four grid columns. When the screen gets narrower and the content is not readable anymore from the squeezed columns, media queries can be used to manipulate the grid styles to show less columns (see Listing 2). [15]

```

1  .grid-item {
2      width: 25%;
3  }
4  @media ( max-width: 768px ) and ( min-width: 421px ) {
5      .grid-item {
6          width: 50%;
7      }
8  }
9  @media ( max-width: 420px ) {
10     .grid-item {
11         width: 100%;
12     }
13 }
```

Listing 2: Example usage of CSS media queries / breakpoints.

There are many relative units in CSS, but *percentage* and *rem* units are chosen to be used across the framework styles. Percentage value is calculated from the relative parent of the element. Rem units act as a multiplier, similar to *em* units, but the rem units are relative to the root element (html-tag) where as the em units are relative to inherited values from the parent. The difference between em and rem units is demonstrated in Listing 3.

```

1  // em
2  html {
3      font-size: 14px;
4  }
5  .grid-item {
6      font-size: 2em;      // 14px * 2 = 28px
7  }
```

```

8  .grid-item p {
9      font-size: 2em;      // 28px * 2 = 56px
10 }
11 .grid-item p span {
12     font-size: 2em;      // 56px * 2 = 112px
13 }
14
15 // rem
16 html {
17     font-size: 14px;      // all rem units relate to this value
18 }
19 .grid-item {
20     font-size: 2rem;      // 14px * 2 = 28px
21 }
22 .grid-item p {
23     font-size: 2rem;      // 14px * 2 = 28px
24 }
25 .grid-item p span {
26     font-size: 2rem;      // 14px * 2 = 28px
27 }

```

Listing 3: Comparing the behavior of em and rem units.

Rem units are easier to control as they relate to the root element and to the root element only, where as the em values can easily get out of hand by inheriting the values multiplying them.

3.6 Browser variation

Different browsers have different *rendering engines* and therefor produce a slightly different rendering results. When one browser supports a new CSS style, others might not. Thus it is important to test the site with multiple browsers. Online service *caniuse.com* has been proven worthy when checking the browsers feature support for example which browsers support SVG images.

It is always good to think of fallbacks to features that are known not to be supported for some browsers. These features can be implemented with additional code to patch-in these missing features. These are called *polyfills* or *shims*. However there should not be a browser or device specific filtering with *user-agent strings* as the method is unreliable and requires constant maintenance. A user-agent string is a list of keywords in the HTTP request header that tell information about the user's browser.

If some of crucial features of the website are not supported by every browser or device, the site should have a *feature detection* implemented. If the user is accessing the site with a browser that does not support all the needed features, user should still be able to access the site and be notified about the issue and how to fix it. For example, the user's browser is in *private browsing* mode, and thus the browser has disabled some of its features. User then tries to access a site that requires one of the disabled features. If the user is not noted about the required feature or not allowed to access the site at all, he or she might think that the site is broken. Alternatively, the site would have a feature detection and the user is allowed to access the site but then notified that the service cannot be used in a private browsing mode. The user then switches to normal browser mode and can use the service as normal.

3.7 Critical Rendering Path

By examining the rendering pipeline of the browser it can be seen how HTML, CSS and JavaScript files end up drawing the pixel on the user's screen. It is important to map and understand the *critical rendering path* of the site as it heavily affects the browsing experience. All starts by generating the HTML *Document Object Model (DOM)* and CSS *Object Model (CSSOM)*. These are visualized in the Figure 16.

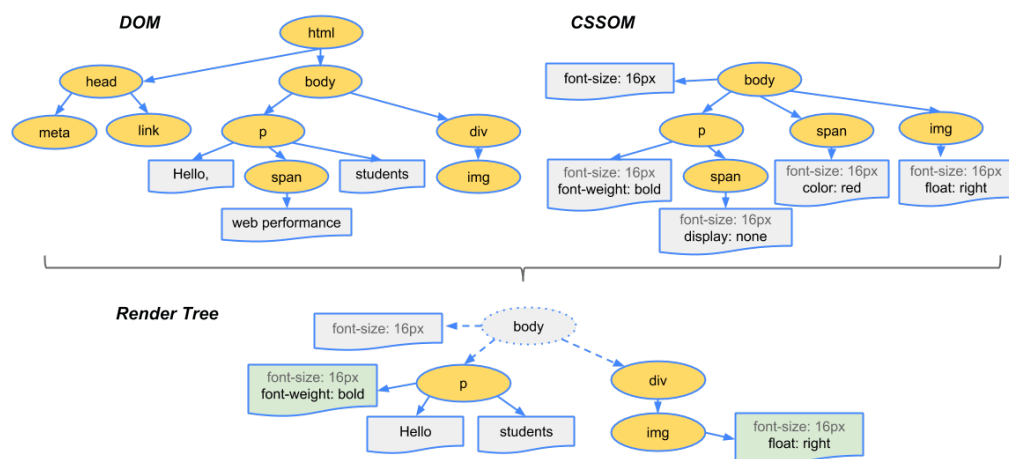


Figure 16: DOM and CSSOM combined into a Render Tree. [7]

When generating DOM from HTML, every tag pair (opening and closing, e.g. `<p>` and `</p>`) creates a node that has properties. Nodes are related to other nodes and together

they create the DOM tree. Similarly with CSS, *rule sets* are converted into nodes that form the CSSOM tree. DOM and CSSOM trees are then combined as a *Render tree* by searching and applying styles for every DOM node from the CSSOM tree. Render tree is then used to generate the *layout* which calculates the exact position and size of each node element with absolute pixel values. Layout is then passed on to the *paint* phase that paints every node as actual pixels on the users screen. This render pipeline is illustrated in Figure 17. [7]

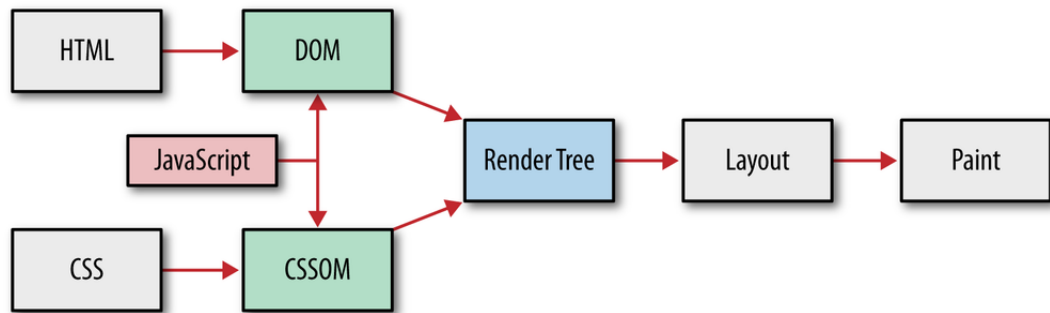


Figure 17: Processing pipeline of a browser. [8]

As seen in the render pipeline, JavaScript can effect HTML and CSS, so it has to be loaded and executed before first paint. This makes the JavaScript a *render blocking resource*. If the JavaScript does not have to be executed before first paint, it should be marked with `defer` or `async` attribute. These attributes will tell the browser that it does not have to wait for these files to be downloaded before rendering the HTML. The `async` attribute makes the JavaScript to be executed when ever it is ready. The `defer` attribute makes the code to be executed after HTML parsing. Figure 18 demonstrates the browser behavior in each case. [7] [8]

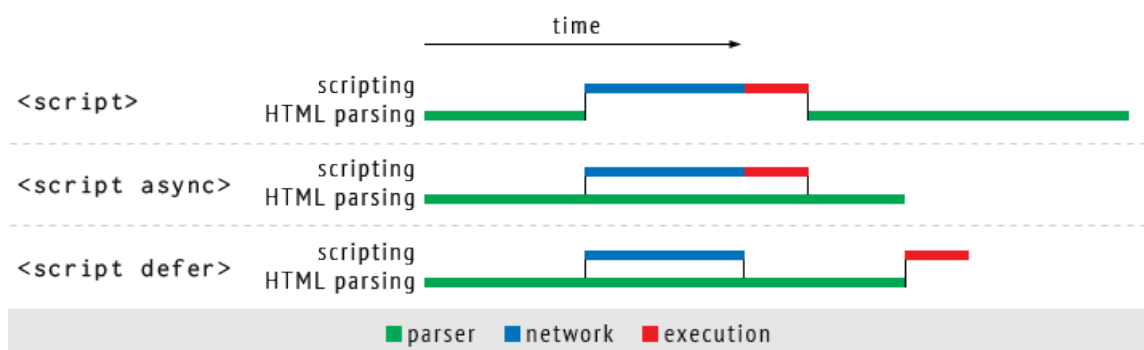


Figure 18: JavaScript loading and execution compared with no additional attributes, with `async` and with `defer` attributes.

Inline JavaScript in the HTML is always parsing-blocking if not deferred by additional code as seen in the first timeline row. When a browser encounters script inside the document

it pauses the DOM construction and hands over the control to JavaScript runtime. [7]

A browser generates styles for every render tree node object by recursively applying the parent styles of the object, from the furthest parent, towards the nearest. Styles are inherited in a cascading manner from parent to child. By default CSS is a render blocking resource. All render blocking resources have to be loaded and processed before browser allows the *first paint* to happen. First paint is the event when the user sees any content. [7]

It is advisable to keep HTML markup and CSS styles separated, but when aiming for the best possible performance, the most critical styles could be provided as *internal styles* or even as *inline styles*. Listing 4 demonstrates the usage of these style types.

```

1  <!DOCTYPE html>
2  <html>
3      <head>
4          <meta charset="utf-8">
5          <meta name="viewport" content="width=device-width, initial-scale=1">
6          <style>
7              body {
8                  background-color: #fefefe;
9              }
10             img {
11                 border: 1px solid black;
12             }
13         </style>
14     </head>
15
16     <body>
17         <div class="above-the-fold">
18             <div class="top-container" style="margin: 0 auto;">
19                 
20                 <h1 style="color: red;">content</h1>
21             </div>
22         </div>
23         <!-- the fold -->
24         <div class="below-the-fold">
25             <p>Here is content below the fold</p>
26             
27         </div>
28         <script src="scripts.js"></script>
29         <script async src="ads.js"></script>
30         <script defer src="foo.js"></script>
31     </body>
32 </html>

```

Listing 4: HTML example.

In Listing 4 there are internal styles in the `header` (on lines 6-13) and inline styles on the `h1` tag (line 20). All JavaScript files are introduced at the bottom of the `body` so they do not interfere with the HTML rendering. The `ads.js` on the line 29 has the `async` attribute to make it be loaded asynchronously as it does not depend on anything on the page.

Internal styles are put into the document `head` block and inline styles are put inside individual element tag as an `style` attribute. By using internal or inline styles, browser does not have to load *external style files* to be able to add styles for the elements as the styles are provided within the HTML file. These critical styles are the ones needed to style the elements above *the fold*. The fold is referring to the position where the content goes out of the viewport - the bottom edge of the screen. The fold is marked as a comment `the fold` in the code listing (line 23). The elements above it have the inline styles but below the fold elements do not.

Above the fold styles can be put to the elements manually or the markup can be generated with additional tools. *Under the fold* styles can be loaded asynchronously. Unlike JavaScript, loading CSS asynchronously is not implemented natively to browsers. Asynchronous CSS loading must be done with JavaScript by adding the style tag to the document *head* after first paint has occurred. [8] [7]

Browsers have a feature called *browser cache*. With it, the browser can re-use once loaded resources. Each resource can define its own cache policy with a *Cache-Control HTTP header*. The HTTP header is essential part of the *HTTP protocol* as it carries information about the transmission. Browser automatically checks for the resources from the local cache. If the resource is not found, it is requested from the server. If the resource is found, it is then validated and then re-used without a need to fetch it again from the server. To validate the files they need to have *cache header* attributes `Cache-Control` and `ETag` in place. In cache-control attribute the `max-age` for the file can be set - it specifies the maximum time in seconds that the response can be reused. ETag is a validation token to check if the resource has been modified after the `max-age` has expired - if not, same file can be used again the same max-age duration. Figure 19 on the next page demonstrates a use case where the browser cache is used. [9]

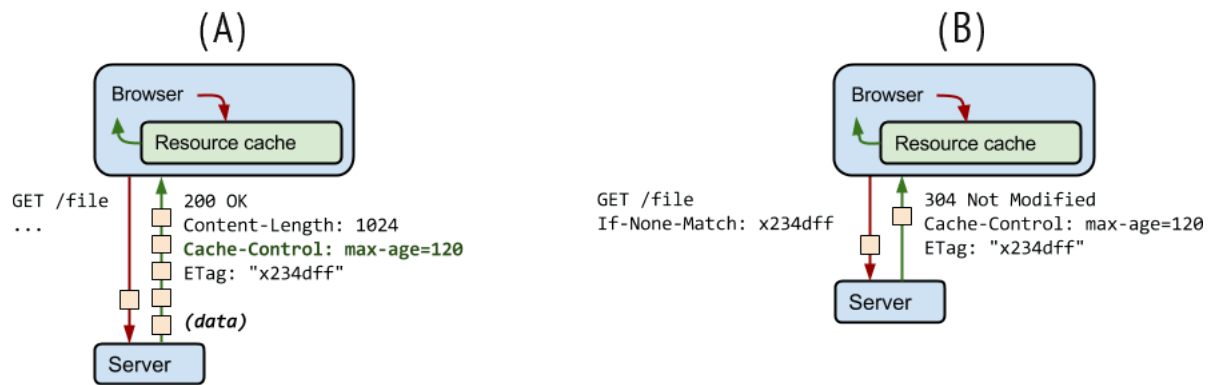


Figure 19: Browser requesting resource from a server. [9]

The *situation A* has the initial resource request in. Server provides the resource with validation ETag and Cache-Control with `max-age=120` which is two minutes. During the two minutes, the browser uses the file straight from the browser cache. In the *situation B*, two minutes has passed and browser requests validation for the resource with the ETag. Resource is not changed so the ETag for the resource stays the same. The server sends the `304 Not Modified` status to indicate that the resource is not changed - the browser now uses the resource for two minutes before asking the server again. The files that does not frequently change, can be set to expire after a long time, e.g. a month or a year. It all depends on how likely it is to that same named resource to be changed. Usually images can be set to expire after a really long time or not at all. HTML, CSS and JavaScript files can be set to expire after a day or a week. The cache profile should be determined independently for each resource.

To invalidate resource from user's browser cache before the max-age of the HTTP request has expired, the url of the resource has to be changed. The resource could have a increasing version number in it is name, and by changing it, the cache is invalidated and browser is forced to load a fresh file.

3.8 Semantics and Search Engine Optimization

Semantics are important when dealing with search engines. Search engines have a ranking system where they give points by certain factors in a website implementation. When

a user performs a search in the search service, it will show the results in relevancy order. The higher points the site has in the search engines indexing, the higher the site is in the search results. The rating algorithms are not public but the search engine providers have guides how to make sites search-engine-friendly. For example Google advises to use meta tags such as `title` and `description` correctly and having urls constructed from words that describes the content. Served resources such as images should have describing filenames. A site gets more points for having links to it in other sites.

When the HTML markup is written for creating the layout, the markup *semantics* should be considered. In HTML every element, attribute and attribute value has a certain meaning. For example headline tags (`<h1>`, `<h2>` ...) should be used with headlines and paragraphs for block of text. If a standard element is available for the content, it should be used for it. When the content is written *semantically correct*, the content can be utilized in many other contexts by *HTML processors*, such as *search engines* or *speech browsers*. [21]

Session length is seen more valuable than just raw *page view* count. When a user is browsing the site and visiting multiple pages, it means that the content is found interesting and valuable for the user. [6]

Search engine robots will crawl through websites and their pages and content. A *site map* can be provided for the search engines, so they can discover pages and site structure of the website more easily. A site map is a XML file that has the site structure information in it. When robots crawl through the content they will look for keywords that are found in meta tags, headers and body text. [6]

4 Framework Development Technologies

This chapter goes through the technologies and techniques used in the web development framework. The key technologies are *node.js* and its package manager *npm*, *gulp.js*, *handlebars.js* and *Sass*. Node.js is a framework for building network applications. Node.js is used to handle the *gulp.js* plugin dependencies with *npm*. Gulp.js is a task automation tool that compiles and optimizes the source files (e.g. *.sass* and *.hbs*) into distribution files (e.g. *.css* and *.html*). [14]

4.1 Templating with Mustache and Handlebars

Templates are preset documents that can be used to generate markup that browsers understand. Templates can be static or dynamic. They can be made dynamic by having variables in the template markup. Template markup is parsed by a template engine that takes the given templates and parameters, replaces the variables by given values and generates the final markup.

Mustache is a logicless templating language. *Handlebars* extends the *Mustache* language by adding more features to it. When templating with *Handlebars* the main advantage is that all template rendering is done with JavaScript. Gulp is built on JavaScript so everything that can be done in JavaScript, can be done in Gulp.

It is not desirable to shift the logic from the back-end to the templates in websites with user specific content. Template logic should be considered as a last resort as processing them with additional logic takes longer. In static sites, like the site of the pilot project, placing the logic into the templates is not an issue as the templates are generated only once by the Gulp and the content served to the users is static HTML. [18]

In case of back-ends applications, the template logic provided by Handlebars is much welcomed as there is no other way to have logical operations when generating HTML from templates. Normally the sites with dynamic content, the HTML generation is done by the parsing services in the back-end or front-end.

In the website of the pilot project there is no content that the user could change or affect, so there is no need for custom server or client side rendering - just static files that are once generated by Gulp.

There are many advantages using templates. Templates help to make the project structure more controllable and maintainable. File structure is easier to visualize when there is a describing module tags representing hundreds of lines of code as seen in Listing 5.

```

1 <html>
2   <head>
3     {{> head}}
4   </head>
5   <body>
6     {{> header}}
7     {{> content}}
8     {{> footer}}
9     {{> javascripts}}
10  </body>
11 </html>

```

Listing 5: Basic Handlebars example.

When presenting a collection of items with identical HTML element structure, templating can be used to avoid manual copying the element structure. Templates take parameters that are then used inside the templates as variables when the list of objects are iterated through. When item element structure changes, these changes need to be made only once to the template where the items are being generated from. Listing 6 is an example of a handlebars template with `for` loops.

```

1 // gallery.hbs
2 <div class='cat-gallery-container'>
3   <h2>Cat gallery</h2>
4   {{#gallery.cat-gallery-items}}
5     <img src='{{src}}' alt='{{alt}}'>
6   {{/gallery.cat-gallery-items}}
7 </div>
8
9 <div class='dog-gallery-container'>
10  <h2>Dog gallery</h2>

```



```

11     {{#gallery.dog-gallery-items}}
12     <img src='{{src}}' alt='{{alt}}'>
13     {{/gallery.dog-gallery-items}}
14 </div>

```

Listing 6: Template for cat-gallery-items and dog-gallery-items JSON data.

The example code iterates through lists of objects from the example *gallery JSON* (Listing 7). These loops are in the lines 4-6 and 11-13. These loops will generate the markup inside them as many times as there are items in the iterated list. The variables inside the loops (`src` and `alt`) are replaced with the values from each objects values in the iterated list.

```

1  // gallery.json
2  {
3      "cat-gallery-items" : [
4          {
5              "src": "images/cat1.jpg",
6              "alt": "Cat in a box."
7          },
8          {
9              "src": "images/cat2.jpg",
10             "alt": "Cat eating."
11          },
12          {
13              "src": "images/cat3.jpg",
14              "alt": "Cat sleeping."
15          }
16      ],
17      "dog-gallery-items" : [
18          {
19              "src": "images/dog1.jpg",
20              "alt": "Dog in a box."
21          },
22          {
23              "src": "images/dog2.jpg",
24              "alt": "Dog eating."
25          },
26          {
27              "src": "images/dog3.jpg",
28              "alt": "Dog sleeping."
29          }
30      ]
31  }

```

Listing 7: List of gallery objects in gallery.json file.

JSON can be used to create objects in JavaScript as the syntaxes are identical. JSON is the best format to be used with Gulp as the Gulp itself runs with JavaScript.

4.2 Styling with Sass

CSS is used to style the content of a HTML document. Style properties and values can be set for elements by targeting them with *selectors*. Properties and values are set in a *declaration* block (see Listing 8).

```

1  body {                                // selector
2                                     // start of declaration block
3      font-family: Helvetica, sans-serif;
4      color: #333;
5                                     // end of declaration block
6  }
```

Listing 8: CSS example.

To achieve a more convenient way of producing CSS, a pre-processor is used to add missing features into the styling language. CSS pre-processors make writing styles more efficient by adding the options to use variables, functions, nesting, inheritance and math calculations. Sass has two syntaxes, the newer SCSS syntax which is more like traditional CSS and the older Sass syntax that uses indentation instead of brackets and semicolons (see Listing 10). In the framework SCSS syntax is preferred. [3]

```

1  // Sass syntax
2  $font: Helvetica, sans-serif
3  $color: #333
4
5  body
6      font-family: $font
7      color: $color
8
9  // SCSS syntax
10 $font: Helvetica, sans-serif;
11 $color: #333;
12
13 body {
14     font-family: $font;
15     color: $color;
16 }
```

Listing 9: Sass/SCSS syntax example.

In the framework, styles are managed with `styles.scss` file that is used to import smaller style files and combining them as one SCSS file to be converted into CSS (see code listing 10).

```
1 // styles.scss
2 @import 'variables';
3 @import 'mixins';
4 @import 'reset';
5 @import 'main';
6 @import 'gallery';
```

Listing 10: SCSS import bundling example.

First Sass variables and useful *mixins* functions are introduced by importing them before other SCSS files. This way the variables and mixins can be used in all other SCSS files imported after them. Mixins are declaration groups that can take parameters. Variables are in `variables.scss` and mixins in `mixins.scss`. The first actual styling file is the `reset.scss` that is used to reset all browser default styles to remove any unwanted styles. Then the rest of the styles can be imported as a logical groups of separate file. The example has some styles divided into `main.scss` and `gallery.scss` files.

4.3 Optimizing Images

Images make up the majority of the overall page size as seen in Figure 20. Over half of the downloaded bytes are for images.

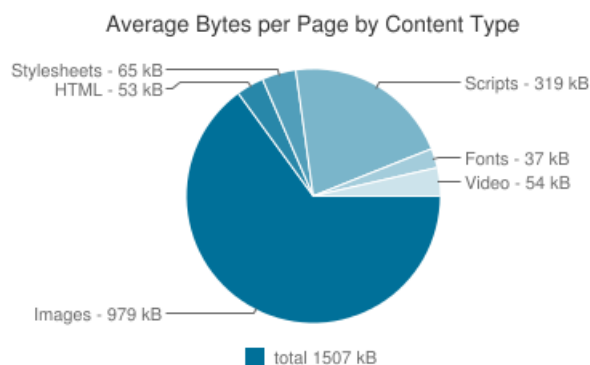


Figure 20: Resource types and the portion of them in top 100 sites. [12]

The task automation tool Gulp can automate some of the optimization process, like creating sprite images, removing unnecessary meta data and compressing images. However,

the most significant decision is made by choosing the right image size and format to save the image.

There are some key factors that differentiate the image formats. The first separation is *vector* and *raster* images. Raster images consist of rectangular pixel grid of color information where as vector graphics are stored as geometric shapes and coordinates. Thus vector images do not lose quality when scaled bigger (see Figure 21). Raster images have only scalability corresponding to the original quality that it was stored with - scaling a raster image up will cause the image quality getting worse. But as the vector images are stored as geometric shapes, they have limited ability to present *photorealistic* images like regular photographs. [23] [10]

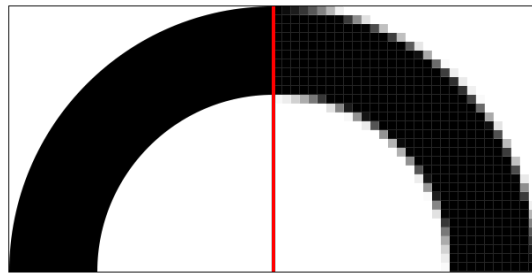


Figure 21: Vector and raster image difference when zoomed in.

There are *lossy* and *lossless* formats of raster images. Lossy format means that the image loses some of its information when the image is saved and compressed. Therefore it is bad to save lossy image multiple times as it loses more of the information of the original image each time it is being saved. Lossless formats do not lose any of the original image information when the image is saved and compressed. [23] [10]

Most used lossy image format would be *JPG* and lossless format *PNG*. The vector image format used in the web is *SVG*. It is important to know when to use what format. If the image consists of simply colored geometric shapes, *SVG* is the best option as it scales infinitely and the file is small. If the image has to perceive the fine details, lossless image format should be picked for the job. If the image has a wide color pallet or partial transparency in it, *PNG-24* is the best pick. If the color pallet is small (less than 256 colors) *PNG-8* can be used. If the image is an ordinary photograph, *JPG* is usually the best format, as it is good with photorealistic pictures and the file size is reasonable even with

larger images. [23] [10]

JPG images can be made *progressive*. This makes the browser first show a low quality version of the loaded image and then incrementally enhance the quality as the image is being downloaded. Important benefit is that user is able to see the image more quickly. The file size of a progressive JPG can be smaller or bigger than the non-progressive equivalent. Under 10 Kilobit images usually are smaller with progressive mode enabled. The gained reduction in file size tends to grow the bigger the file size is. Same incremental enhancement can be done with PNG images by saving them with option *interlaced* enabled. Interlaced PNGs are a bit bigger but the gained user experience make this a good trade-off. [23] [10]

Devices with high pixel density screens have capability to show more sharper images. To get the benefit from this high resolution screen, high resolution images are also needed. However, as the resolution goes up, so does the file size as there are more information to be stored with more pixels. With high pixel density devices there are less *CSS pixels* for each actual device screen pixel (see Figure 22). This is done so the content would not shrink too much to be unreadable.

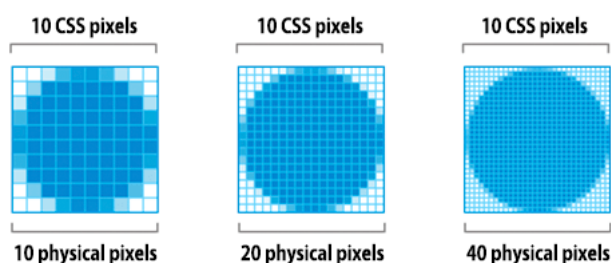


Figure 22: 1:1, 2:1, 4:1 device/CSS pixel ratio. [10]

If there are images on the page that are not needed right away, *lazy-load* functionality can be added to them with JavaScript. This way the browser has less files to load when arriving to the site and the page rendered faster. This also saves bandwidth and the available connections for resources that are more important. Listing 11 has an example of lazy-loading image.

```
1 
```

Listing 11: Example image markup with lazy-load.

The image has an generic spinner gif as a placeholder image and a “*spinner*” alt placeholder text. There are also extra attributes `data-src` and `data-alt` that have the actual image information in them. When the page is being loaded, the lazy-load image is shown as an animated spinner gif. When the page is fully loaded and rendered, there is a piece of JavaScript code that goes through all the images with these additional data-attributes and replaces the default `src` and `alt` values as the ones found in the data-attributes. Alternatively the lazy-load functionality can be triggered for each image only when they are visible on the browsers viewport.

4.4 Modularity with Package Managing

The key for making future proof framework is to make it as modular as possible - that way any part of the framework can be changed to a similar solution. By using code packages from npm for building the framework, the architecture of the framework is more clear and understandable. With this clear and understandable structure, the framework can be shaped around the project and not the other way around.

In the most optimal situation, when a feature specification of a new application is written down, every component in it is already once implemented and can be found in the web development framework with its configurations. So all it would take would be to clone the framework as a base for the project and configure it with the project specific information.

If the needed feature has not been implemented into the framework, the implementation solutions of the feature should be researched. After research the best suiting, and usually most popular, package from *Node package manager* (npm) can be added to the project. The npm is a tool to share reusable JavaScript code between developers. The npm provides a vast selection of components with the Unix philosophy: “*do one thing and do it well*”. The idea behind this is to use small building blocks of reusable code *packages* to construct a custom solution for any project. Although the stack of components needed in different projects is not exactly the same, they all use some common packages. This is where the strength of the web development framework lies. As the npm manages the

packages, npm can be told to fetch available updates for the packages. npm provides the reusable code packages and the web development framework has them implemented. [20]

4.5 JavaScript Scoping Convention

In JavaScript everything runs in a *global scope* by default. This can lead to naming collisions and inefficient memory usage as there are no *block scopes* like in the other common programming languages. *Scope* refers to the current context of the code - the currently usable variables and functions in the specific block of code. In other common programming languages, *block statements* create a block scope. Blocks are formed with a pair of curly brackets ({ and }) like found in `if` or `for` statements.

In JavaScript, a new function always creates a new scope. When a scope is created inside another scope, the nested one can access the scope of the parent and use its variables and functions. This is called *lexical scope*. To manage scopes without having the ordinary block scoping feature, *modules* are used. Modules have an interface but the implementation and state of the module are hidden. Module is created by using an *object* or a *function*.

```

1  var theObject = {
2      publicVariable:null,
3      publicFunction:function(){ }
4  }
5
6  theObject.publicFunction();

```

Listing 12: Putting variables and functions into an object separates them from the global scope.

All the functions and variables of the object in Listing 12 are available in public scope. The properties of object can accessed with *dot notation* (`Object.property`) or square brackets (`Object[property]`).

```

1  var theModule = function(){
2      var publicVariable = null;
3      var _privateVariable = null;
4      function publicFunction(){ }

```

```

5     function _privateFunction(){ }
6
7     return {
8         publicVariable:publicVariable,
9         publicFunction:publicFunction
10    }
11 }();
12
13 theModule.publicFunction();

```

Listing 13: Anonymous function creates a new scope and separates the variables in it from the global scope. Returned object acts as an interface.

As seen in Listing 13 on the preceding page, wrapping the code block into an anonymous function, variables and functions inside it are not available to outside. All functions wanted to be exposed can be put to the returned object that acts as an interface to the closed scope. From outside, `publicFunction` would be called like `theModule.publicFunction()`. To differentiate the private functions and variables from public, underscore can be added in front of the name (see Listing 13 on the previous page, lines 3 and 5).

At the time of writing the study, the sixth edition of ECMAScript, the new JavaScript standard, support is very limited in browsers. The new standard is called ECMAScript 2015 or ECMAScript 6 (referred as *ES6*). With the ES6, the JavaScript language is made more mature and easier to write bigger projects with. ES6 borrows some much needed features from other programming languages. Some of the most notable are *classes*, variable types *let* and *const*, *arrow functions* and *promises*. [22]

4.6 Back-end

If the content served for the clients needs to be dynamic, back-end is needed. But the application itself does not need to have a back-end built into it - the application can use a feed to parse the dynamic content in client side. Dynamic content can be fetched from external source, from *API* or *feed*, or the content can be created internally in *CMS*. *API* is a interface to request content from a back-end application. *Feed* is a static file served and updated by the back-end that the client application reads.

At the time of writing this thesis, the framework did not have a ready back-end implementation thought out. The pilot project of the framework is a back-endless website and content is in a one HTML file, although there are multiple subpages. All the content markup is already loaded in the HTML file when user loads it, but some of the content is hid. The visible section is determined by the *url slug* and the section switching is done with JavaScript and CSS. As there are no additional pages, there are configurations done in the server side so all the traffic were directed to the index page.

5 Developer's Workflow with Framework

At the time writing this thesis the framework itself was not separated from the pilot project to be its own project. So if a new site were created using the framework, it would be done on top of the pilot project.

New tools and practices will be added to the framework as they are needed and discovered. There were a couple of practices discovered and well-proven from previous projects. For version control the framework uses *Git* with *Git Flow* convention, for styling *SASS guideline* and with JavaScript *object oriented javascript*. When there are named conventions for a project, it is easier for multiple people to work on the project together. When the work steps are clear and predetermined it is easier to measure time spent in the workflow steps. After a few iterations it is easier to do estimates how long each step would take. Estimates are especially valuable for paying customers as they usually need to approve a budget internally before the project can be started.

5.1 Framework structure

Main structure of the web development framework is divided into a source folder *src* and to a distribution folder *dist* as seen in Figure 23 on the following page. The *src* folder contains all the files used to develop the website. There are then own folders for the different file types to keep everything nice and organized. *Dist* folder is the place where the files are compiled from the source files.

At the root of the folder structure there are also folders `.git`, `build`, `node_modules`, generated by git and npm. They do not contain files that developer should touch. Folder root also contains `gulpfile.js` and `package.json` files. The `gulpfile.js` is used by Gulp and it contains all the automation tasks. The npm uses the `package.json` to determine

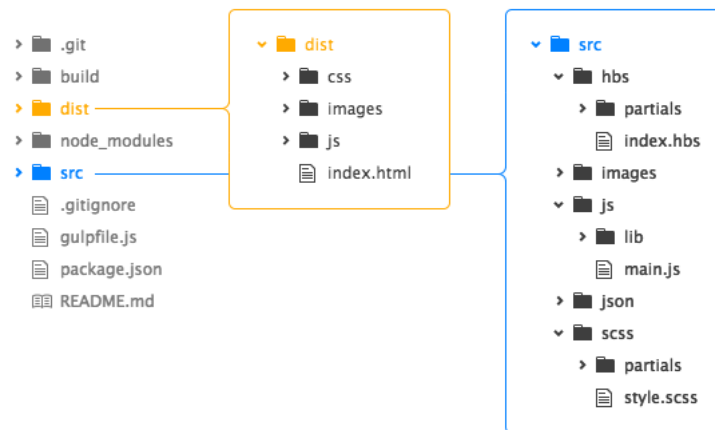


Figure 23: Frameworks basic folder structure.

all the project dependencies. The developer just needs to type command `npm install` into the command line, to install all project dependencies.

5.2 Version Control System

Version control system (VCS) is a tool that records changes made to the project files. Version control is essential tool for bigger projects to keep track what has been done to where, when it was done and who did it. If version control is done right it is like having a back-up after each change made to the project. Every change to the project is incrementally added to previous version. So rather than taking a hard copy of all the files as in their current state, only changes are saved. This technique saves a lot of space and makes version comparison easier. Version control also allows a team work with same files at the same time.

In the web development framework version control is done by using *Git* with addition of online service *GitHub*. GitHub is a web service to store, view and manage Git projects. Git was chosen because of the previous experience using it. Additionally Git is the most used version control system at the moment and as a result of that it has a large community support.

Git and most other version control systems can be used from command line or with graphi-

cal interface. Command line is a powerful tool but when comparing the difference between versions it's better to see the files visually in a text editor. [5]

A simplified version of the *Gitflow* has been proven to be an excellent way of using git. In this simple Gitflow model there are a *master*, *development* and *feature* branches. The master is for the stable production-ready code. The development branch is for merging and gathering features. The feature branches are for developing new features. An example of this is seen in Figure 24. [1] [4]

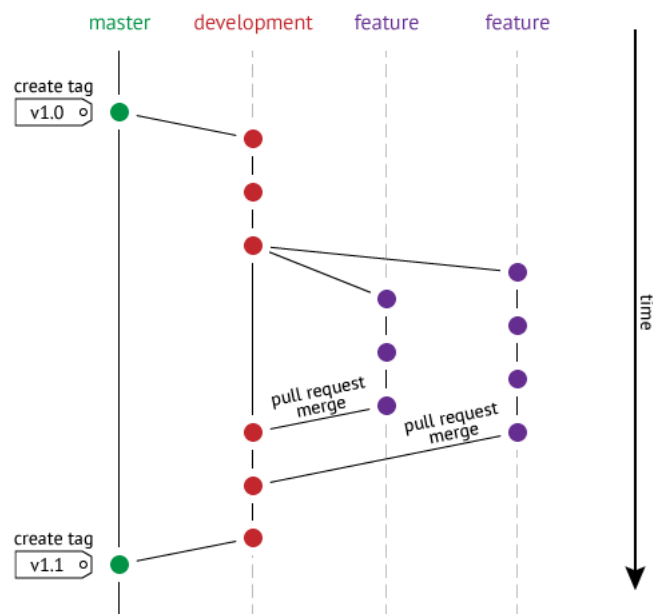


Figure 24: Gitflow model.

When a new set of features are to be done, development branch should be up-to-date with master. Smaller changes can be done straight to the development branch but bigger features should be forked to their own branches from development branch. When the feature is ready it is merged back to the development branch. This can be done by *pull request* or just plain merge. Pull request is a GitHub feature where two branches diff can be examined. Pull request generates an overview of the proposed merge. The pull request can be assigned to others and comments can be written.

When all the features are merged to the development branch, these changes can be put to the master branch. At this stage, a *release tag* should be created of the current state of the master branch. Git tags are used when a point in the commit history is important.

5.3 Browser Developer Tools

Browser developer tools are essential when developing a web site as they provide a way to see what is happening in the browser while the site is being used. Figure 25 shows some of the Chrome's developer tools.

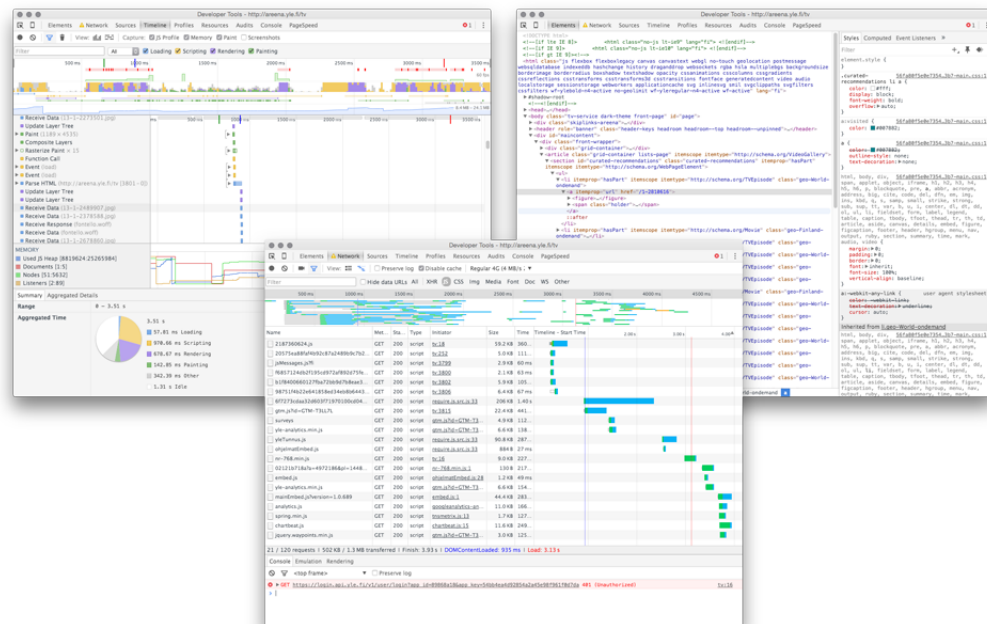


Figure 25: Google Chrome browser development tools.

In the *Elements* tab, the webpage structure and styles can be manipulated in real-time. Network panel offers a visual *waterfall* timeline graph of the requested and downloaded files. The details of the requests can also be closely examined.

Console acts as a way to live code JavaScript. It is also possible to put *break-points* in the middle of JavaScript code to pause the execution on it when it is reached. While the code execution is paused, the current scope of the program can be investigated and interacted with through console. The code can be executed one command at the time and while keeping track of the execution from the source files in the *Sources* tab. There is also the *Timeline* tab for recording and exploring the events happening in the browser during the page markup parsing and code execution. *Resource* tab provides an interface to explore the browsers local resources such as *cookies* or *local* and *session* storage.

Render performance can be investigated with *fps*, *memory* and *processor* usage monitors and *layer highlighting*. Layer highlighting provides a way to see live, which layers of the layout are being repainted while the page is being interacted with.

Chrome also provides a way to remote debug a webpage from a connected Android device. The page in the device can be interacted through the developer tools the same way as the normal desktop browser.

5.4 Task Automatization with Gulp

Task runners help to automate tedious manual work such as writing compile commands into a console. Most commonly used front-end task runners are Gulp and Grunt. Gulp is more versatile coding-like tool while Grunt is more configuration-based with built-in common tasks. Both of these task runners support custom plugins.

Web development framework uses *Gulp*. It acts as a center point for the whole framework. Gulp, with its plugins, makes it possible to develop with continuous integration practice. Gulp's functionality constructs of tasks that are sets of instructions. These instructions tell Gulp and its plugins what to do in what situation. Most of the time a task includes taking a file, compiling it and then writing the result into a file. For example SCSS is compiled into CSS and moved to a dist-folder. Handlebars are compiled into HTML and moved to dist-folder. On top of that Gulp minifies and optimizes the files before putting them into the dist-folder.

Gulp offers tools such as CSS preprocessing, JavaScript minifying, live-reload. One of the main advantages of developing with Gulp is that every modification is instantly visible. This is made possible with Gulp *watch* functionality. Gulp can be set to observe files and folders and when a change to them is made, Gulp can trigger a *task*.

With task automation and *Browsersync* it is easy to demo the current version of website

and gather feedback. Browsersync provides an easy way for a developer to have a light weight server running in the local computer. This makes the web page available in the local area network, so anyone in the same local network can view the website. Listing 14 presents a short clip from the framework `gulpfile.js` where all of the code for Gulp tasks is located.

```

1 // gulpfile.js
2 'use strict';
3
4 var gulp = require('gulp');
5 var hbs = require('gulp-hb');
6
7 gulp.task('default', ['clean'], function (cb) {
8     runSequence('hbs', 'browser-sync', 'watch', cb);
9 });
10
11 gulp.task('hbs', function() {
12     return gulp.src('src/hbs/index.hbs')
13         .pipe(hbs({
14             debug: true,
15             data: './src/json/**/*.json',
16             helpers: './src/helpers/*.js',
17             partials: './src/hbs/partials/**/*.hbs',
18             bustCache: true
19         })))
20         .pipe(rename('index.html'))
21         .pipe(minifyHtml())
22         .pipe(gulp.dest('dist'));
23 });
24
25 gulp.task('watch', function() {
26     gulp.watch('src/hbs/**/*.hbs', ['hbs', browserSync.reload]);
27 });

```

Listing 14: Example Gulp task for generating HTML from hbs files.

On top of the file, in line 4, required plugins are declared and assigned into variables to be used in the tasks. In the line 11 the “hbs” task is created. In it, the `index.hbs` file is taken and piped through hbs plugin that returns HTML generated from the hbs files. The result is piped forward and renamed, minified and moved to the “dist” folder.

In the line 7 *default* task is created. It is the task that is being called by default when Gulp is started from the command line. Second parameter in the default task is a dependency for the “clean” task. Dependencies are ran before the other tasks declared in the `runSequence` of the default task. Last task before the callback (`cb`) is the “watch” task that will leave the gulp running and monitoring the `src/hbs` folder. If there are any changes in

the hbs folder, Gulp will run the “hbs” task where it recompiles all the hbs files into HTML and the triggers the browser reload with the `browserSync.reload` command.

6 Summary

The framework created in the present study can be thought of as a success. At the time of writing the thesis, there were two websites made with the framework. The second project after the pilot project was made in one evening - in few hours it was designed, implemented and put to production. The gathered tools and conventions that form the framework have proven themselves to be an effective way to develop websites.

The framework needs to be separated from the pilot project into its own repository where it can be further developed and cloned without having unnecessary files from other projects. The framework modules and libraries should be separated to some kind of a packet managing system. This way the libraries could be updated to every project without a need to refactor them separately for every project. Libraries could be updated manually by copying and replacing the old file or preferably some sort of package manager could be implemented to handle the updating. All the project specific code for libraries would be in a configuration file.

To further develop the framework, some extra steps need to be added to the normal development cycle. For example during the implementation of the pilot projects, the featured item carousel and photo viewer were implemented in a generic manner so that the code can be more easily reused. This usually means separating the feature into its own code module and abstracting the code so it relays on configuration and not on hard-coded values.

The features and structure of the framework should be documented so it would be easier to introduce it to other developers. The framework would also need a name if it were to be advertised and sold as a service.

Different kinds of back-end solutions should be studied to be able to have dynamic con-

tent in the sites. Server-side conventions, for example logging and monitoring server activity, are not yet researched. If the framework was to be used with customer projects, the server-side aspect of the web application needs to be studied. This would open the possibility to create dynamic content with APIs and feeds. Also some kind of a content management system would needed to be implemented to create the content for the APIs and feeds. One big features would be user accounts and money handling. These would make it possible to develop web stores.

References

- [1] Vincent Driessen. Git Branching. URL: <http://nvie.com/posts/a-successful-git-branching-model/> (visited on 04/26/2015).
- [2] Maximiliano Firtman. High Performance Mobile Web. O'Reilly Media, Inc. URL: <https://www.safaribooksonline.com/library/view/high-performance-mobile/9781491912546/ch02.html> (visited on 11/10/2015). digital book, early release (not finished).
- [3] Hugo Giraudel. SASS Guidelines. URL: <http://sass-guidelin.es> (visited on 04/26/2015).
- [4] Git. Git Flow. URL: <https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow> (visited on 04/26/2015).
- [5] Git Getting Started. URL: <http://git-scm.com/book/en/v2/Getting-Started-About-Version-Control> (visited on 04/26/2015).
- [6] Google. Google's Search Engine Optimization Starter Guide. URL: <http://static.googleusercontent.com/media/www.google.com/fi//webmasters/docs/search-engine-optimization-starter-guide.pdf> (visited on 04/26/2015).
- [7] Ilya Grigorik. Analyzing Critical Rendering Path Performance — Web Fundamentals. URL: <https://developers.google.com/web/fundamentals/performance/critical-rendering-path/analyzing-crp> (visited on 10/03/2015).
- [8] Ilya Grigorik. High Performance Browser Networking. O'Reilly Media, Inc. URL: <https://www.safaribooksonline.com/library/view/high-performance-browser/9781449344757> (visited on 10/03/2015). digital book.
- [9] Ilya Grigorik. HTTP Caching. URL: <https://developers.google.com/web/fundamentals/performance/optimizing-content-efficiency/http-caching> (visited on 11/19/2015).
- [10] Ilya Grigorik. Image optimization. URL: <https://developers.google.com/web/fundamentals/performance/optimizing-content-efficiency/image-optimization?hl=en> (visited on 11/20/2015).

- [11] Lara Callender Hogan. Designing for Performance. O'Reilly Media, Inc. URL: <https://www.safaribooksonline.com/library/view/designing-for-performance/9781491903704/ch03.html> (visited on 11/08/2015). digital book.
- [12] HTTP Archive - Interesting Stats. URL: <http://httparchive.org/interesting.php> (visited on 11/15/2015).
- [13] Jeff Johnson. Designing with the Mind in Mind, 2nd Edition. Morgan Kaufmann. URL: <https://www.safaribooksonline.com/library/view/designing-with-the/9780124079144/xhtml/CHP002.html> (visited on 11/08/2015). digital book.
- [14] Inc. Joyent. About Node.js. URL: <https://nodejs.org/en/about/> (visited on 11/30/2015).
- [15] Florian Rivoal; Håkon Wium Lie; Tantek Çelik; Daniel Glazman; Anne van Kesteren. Media Queries. URL: <http://www.w3.org/TR/css3-mediaqueries/> (visited on 11/13/2015).
- [16] Peter-Paul Koch. A tale of two viewports. URL: <http://www.quirksmode.org/mobile/viewports.html> (visited on 11/11/2015).
- [17] Travis Lowdermilk. User-Centered Design. O'Reilly Media, Inc. URL: <https://www.safaribooksonline.com/library/view/user-centered-design/9781449359812/ch02.html> (visited on 11/07/2015). digital book.
- [18] Gabriel Manricks. Instant Handlebars.js. Packt Publishing. URL: <https://www.safaribooksonline.com/library/view/instant-handlebarsjs/9781783282654/> (visited on 11/14/2015). digital book.
- [19] Jakob Nielsen. 10 Usability Heuristics for User Interface Design. URL: <http://www.nngroup.com/articles/ten-usability-heuristics/> (visited on 11/05/2015).
- [20] Inc. npm. What is npm? URL: <https://docs.npmjs.com/getting-started/what-is-npm> (visited on 11/01/2015).
- [21] Ian Hickson; Robin Berjon; Steve Faulkner; Travis Leithead; Erika Doyle Navara; Edward O'Connor; Silvia Pfeiffer. W3C - HTML5 - 3 Semantics, structure, and APIs of HTML documents. URL: <http://www.w3.org/TR/html5/dom.html> (visited on 09/02/2015). 3.2.1 Semantics.
- [22] Narayan Prusty. Learning ECMAScript 6. Packt Publishing. URL: <https://www.safaribooksonline.com/library/view/learning-ecmascript-6/9781785884443/> (visited on 11/15/2015). digital book.

- [23] Steve Souders. Even Faster Web Sites. O'Reilly Media, Inc. URL: <https://www.safaribooksonline.com/library/view/even-faster-web/9780596803773/ch10.html> (visited on 11/20/2015). digital book.
- [24] StatCounter Global Stats. URL: <http://gs.statcounter.com/#all-comparison-ww-monthly-200812-201510> (visited on 11/14/2015).
- [25] Ben Caldwell; Michael Cooper; Loretta Guarino Reid; Gregg Vanderheiden; Wendy Chisholm; John Slatin; Jason White. Web Content Accessibility Guidelines (WCAG) 2.0. URL: <http://www.w3.org/TR/WCAG20/> (visited on 11/11/2015).